Knowledge Representation and Inference

2019/2020

©Luís Seabra Lopes

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Last update: 2019-10-20

I Objectives

The present list of exercices focuses on knowledge representation and associated inference mechanisms. Work is based on a module with simple implementation of semantic networks. Students will extend the initial version of the module with new functionalities. Recall now that a semantic network represents knowledge in the form of a graph, where nodes represent entities (objects or types) and edges represent relations between entities. We will also use a Bayesian network module.

This list of exercices is used in the *Artificial Intelligence*, (*Licenciatura em Engenharia Informática*) and *Introduction to Artificial Intelligence* (Mestrado Integrado em Engenharia de Computadores e Telemática) courses.

This work will be done in 4 to 5 practical classes. ***For a fruitful use of the classes, exercises that are within the thematic scope of a given class should be completed before the next class.***

II Semantic Networks

1 The initial module

O módulo semantic_network, fornecido em anexo, exporta um conjunto de classes para representar relações semânticas:

The semantic_network module exports a set of classes to represent semantic relations:

```
class Relation:
    def __init__(self,e1,rel,e2):
        self.entity1 = e1
        self.name = rel
        self.entity2 = e2

class Association(Relation):
```

```
    def __init__(self,e1,assoc,e2):
        Relation.__init__(self,e1,assoc,e2)

class Subtype(Relation):
    def __init__(self,sub,super):
        Relation.__init__(self,sub,"subtype",super)

class Member(Relation):
    def __init__(self,obj,type):
        Relation.__init__(self,obj,"member",type)
```

Using these classes, we can represent generic associations (class `Association`) generalization relations (class `Subtype`) and membership relations between objects to their respective types (class `member`).

The relations of `subtype` and `member` allow inheritance of properties.

Examples:

```
>>> a = Association('socrates','professor','filosofia')
>>> s = Subtype('homem','mamifero')
>>> m = Member('socrates','homem')
>>> str(a)
'professor(socrates,filosofia)'
>>> str(s)
'subtype(homem,mamifero)'
>>> str(m)
'member(socrates,homem)'
```

Of course, a typical application of a semantic network is its use to represent the knowledge of an agent. The agent may obtain knowledge from various sources, including human interlocutors (users) with whom it interacts. With this module, we can associate the users to the relations they declare, using the following class:

```
class Declaration:
    def __init__(self,user,rel):
        self.user = user
        self.relation = rel
```

Here, we use the class `Declaration` to record that `user` declared the relation `rel`.

Examples:

```
>>> da = Declaration('descartes',a)
>>> ds = Declaration('darwin',s)
>>> dm = Declaration('descartes',m)

>>> str(da)
'decl(descartes,professor(socrates,filosofia))'
>>> str(ds)
'decl(darwin,subtype(homem,mamifero))'
>>> str(dm)
'decl(descartes,member(socrates,homem))'
```

Finally, the class `SemanticNetwork` represents a semantic network using a list of declarations of relations:

```
class SemanticNetwork:
    def __init__(self):
        self.declarations = []
    def __str__(self):
        pass
    def insert(self,decl):
        self.declarations.append(decl)
    def query_local(self,user=None,e1=None,rel=None,e2=None):
        self.query_result = \
            [ d \
                    for d in self.declarations \
                    if  (user == None or d.user==user) \
                    and (e1 == None or d.relation.entity1 == e1) \
                    and (rel == None or d.relation.name == rel) \
                    and (e2 == None or d.relation.entity2 == e2) ]
        return self.query_result
    def show_query_result(self):
        pass
```

The function `query_local` allows to obtain local information (that is, not inherited) about the entities present on the network. This function can be parameterized by the user (`user`), the name of the first entity involved in the relation (`e1`), the relationship name (`rel`) and the name of the second entity involved in the relation (`e2`). The function will return all declarations that satisfy specified parameters, some of which may be omitted.

Example of creating and querying a semantic network:

```
>>> z = SemanticNetwork()
>>> z.insert(da)
>>> z.insert(ds)
>>> z.insert(dm)
>>> z.insert(Declaration('darwin',Association('mamifero','mamar','sim')))
>>> z.insert(Declaration('darwin',Association('homem','gosta','carne')))
>>> z.insert(Declaration('descartes', Member('platao','homem')))
>>> z.query_local(user='descartes',rel='member')
........
>>> z.show_query_result()
decl(descartes,member(socrates,homem))
decl(descartes,member(platao,homem))
>>>
```

2 Exercises

The module described above is generic but contains some limitations. It would be interesting to implement some additional functionality. For testing the new features, you can use the module sn_example, which contains an example of a semantic network with several declarations already introduced. The examples given in some of the exercices refer to the contents of this module.

1. Develop a function that returns the list (of names) of existing associations.
2. Develop a function that returns the list of objects whose existence can be inferred from the network, that is, a list of entities declared as instances of some type.
3. Develop a function that returns the list of existing users on the network.

4. Develop a function that returns the list of types on the network.

5. Develop a function that, given a entity, returns the list (of the names) of the locally declared associations.

6. Develop a function that, given a user, returns the list (of the names) of the relations declared by the user

7. Develop a function that, given a user, returns the number of different associations used in the relations s/he has declared.

8. Develop a function that, given an entity, returns a list of tuples, in which each tuple contains (the name of) a locally declared association and the user who declared it.

9. An entity `A` is predecessor (or ancestor) of an entity `B` if there is a chain of `member` and/or `Subtype` relations connecting `B` to `A`. Develop a function that, given two entities (two types, or one type and one object), return `True` if the first is the predecessor of the second, and `False` otherwise.

```
>>> z.predecessor('vertebrado','socrates')
True
>>>
>>> z.predecessor('vertebrado','filosofo')
False
>>>
```

10. Develop a function that, given two entities (two types, or a type and an object), where the first is the predecessor of the second, return the list of entities found on the path from the first to the second entity. If the first entity is not a predecessor of the second entity, the function returns `None`.

```
>>> z.predecessor_path('vertebrado','socrates')
['vertebrado','mamifero','homem','socrates']
>>>
```

11. The function query_local() does not return inherited knowledge. It only returns local declarations for a given entity.

    (a) Develop a new function query() in class SemanticNetwork that allows to obtain a list with all local and inherited declarations for a given entity. The function receives as input the entity and, optionally, the name of the association.

```
>>> z.query('socrates','altura')
.....
>>> z.show_query_result()
decl(descartes,altura(mamifero,1.2))
decl(descartes,altura(homem,1.75))
decl(simao,altura(homem,1.85))
decl(darwin,altura(homem,1.75))
```

    (b) Develop a new function query2() in class SemanticNetwork that returns all the local declarations (including `Member` and `Subtype`) as well as inherited declarations (only `Association`) in an entity. The function receives as input the entity and, optionally, the name of a relation. (Note: You can use the function of the previous exercice to build part of the result.)

12. Develop a new query function, query_cancel(), similar to the query() function, but in which there is cancellation of inheritance. In this case, when an association is declared in an entity, the entity will not inherit this association from the predecessor entities. The function receives as input the entity and the name of the association.

13. Develop a function query_down() in class SemanticNetwork that, given a type and (the name of) an association, please return a list with all statements of this association in descendant entities.

14. Sometimes, in the absence of known general information, it may be useful to use inductive inference. In this case, information on more specific entities (subtypes and / or instances) can be used to infer general properties of a type. Develop a function query_induce() that, given a type and (the name of) an association, returns the most frequent value in the descending entities.

15. So far, we have assumed that properties (associations) can have multiple values in a given entity. For example, the association gosta (likes) can have several values (you can like fish and meat).

    In a professional system, there are other types of associations. There are associations that admit only one value. For example, father admits only one value (the father of the person).

    In the case of associations with numeric value, for example altura (height) or peso (weight), the existence of several values can be treated as a distribution, where the average of these values can be used as a reasonable approximation to the truth.

    To test the following two exercices, uncomment the statements which are commented out in the sn_example module.

    (a) Develop two classes derived from class Relation to represent associations with a single value (class AssocOne) and associations with numerical values (class AssocNum).

    (b) Develop a new function query_local_assoc() in class SemanticNetwork, which allows querying values of the local associations of a given entity, taking into acount the different types of associations as follows:

    - Association - Returns a list of pairs $(val, freq)$ with the most frequent local values and respective frequencies. To select of the most frequent values, values are added to the list by decreasing order of its frequency until the sum of the frequencies reach a value equal or higher than 0.75.
    - AssocOne - Returns a pair $(val, freq)$, where val is the most frequent local value, and $freq$ is the frequency (percentage) with what occurs.
    - AssocNum - Returns the average of local values.

    The function receives an entity and (the name of) an association, and returns the result as described above.

    Exemplos:

```
>>> z.query_local_assoc('socrates','pai')
('sofronisco', 0.67)
>>> z.query_local_assoc('socrates','pulsacao')
56
>>> z.query_local_assoc('homem','gosta')
[('carne', 0.40), ('peixe', 0.40)]
>>>
```

16. In this type of semantic network, given the accumulation of declarations of multiple interlocutors, inconsistencies may occur in the stored knowledge. When there is a conflict between the values assigned to an association within an entity, it makes sense to take into acount the inherited values. Develop a query_assoc_value() whereas given an entity, E, and (the name of) an association, A, return the value, V, of this association in this entity, according to the following case analysis:

    - If all local declarations of A in E assign the same value, V, to the association, then that value is returned, ignoring the values possibly declared for te predecessors
    - Otherwise, return the value, V, which maximizes the following function:

      $$F(E, A, V) = \frac{L(E,A,V)+H(E,A,V)}{2}$$

      where $L(E, A, V)$ is the percentage of declarations of V for A in the entity E, and $H(E, A, V)$ is the percentage of similar statements in the predecessor entities of E.
    - If the association does not exist in predecessors, the function returns the most frequent value, that is, the value that maximizes $L(E, A, V)$.

    In this exercice, you should ignore the association types introduced in the previous exercice (i.e. `AssocOne` and `AssocNum`).

III Bayesian Networks

1 Presentation of the initial module

The bayes_net module, attached hereto, exports a class to represent Bayes networks (BayesNet). A method is already implemented joint_prob(conjunction) to calculate the joint probability. The bn_example has the 'alarm' example of theoretical classes.

2 Exercises

1. Create a new Bayes network to represent the knowledge given in exercise III.11 of the *Theoretical-Practical Guide*.
2. Develop a new method that, given a network variable and a Boolean value, calculate its probability individual.
3. Using the network from the previous paragraph, calculate the probability of a user needing help.